CHAPTER 5
I/O DEVICE DEVELOPMENT ENVIRONMENT

In Chapter 4 we learned that an I/O device consists of a USB transceiver, a Serial Interface Engine (SIE), a protocol controller (typically a microcontroller), and interfaces to the real world. The microcontroller will also run an application program that implements the functionality of the I/O device. The I/O device will communicate with the PC Host using standard USB protocols and they will exchange data blocks. A program on the PC Host will be responsible for coordinating the actions of the I/O device. Wow! A typical USB application involves a lot of software and running on two processors. Fortunately the development process is well defined and there are many tools available. Figure 5-1 shows the tasks that must be completed to implement a working USB I/O device design – some of the tasks have already been implemented so we need only focus on the unique aspects of our I/O device design.

Figure 5-1. The tasks needed for a USB I/O device design

It should be stressed at this time that the PC host and I/O device communicate using standard protocols – this means that the implementation of the I/O device is unknown to the PC host. The software on the PC host does not change however we implement the I/O device. This separation of tasks gives us some design freedom – we could implement an easier, over-featured I/O device as a prototype then later cost reduce the solution. This will be the strategy I shall use in Chapter

7 – we'll focus on functionality then we'll cost reduce in Chapter 9. The I/O device hardware and firmware can be developed independently of the PC host software and in this chapter we'll start with the I/O device hardware.

The choice of I/O device hardware is large, and the development task will be specific to the solution chosen. This chapter starts at a high general level and then describes different groups of solutions using a representative example from each group.

DEVELOPMENT TOOLS

In this section the I/O device that we are developing is called the **target** and the PC platform is called a development **host**—because it will host all of our development tools. Figure 5-2 shows an ideal development environment where the target includes its own USB-connected PC platform.

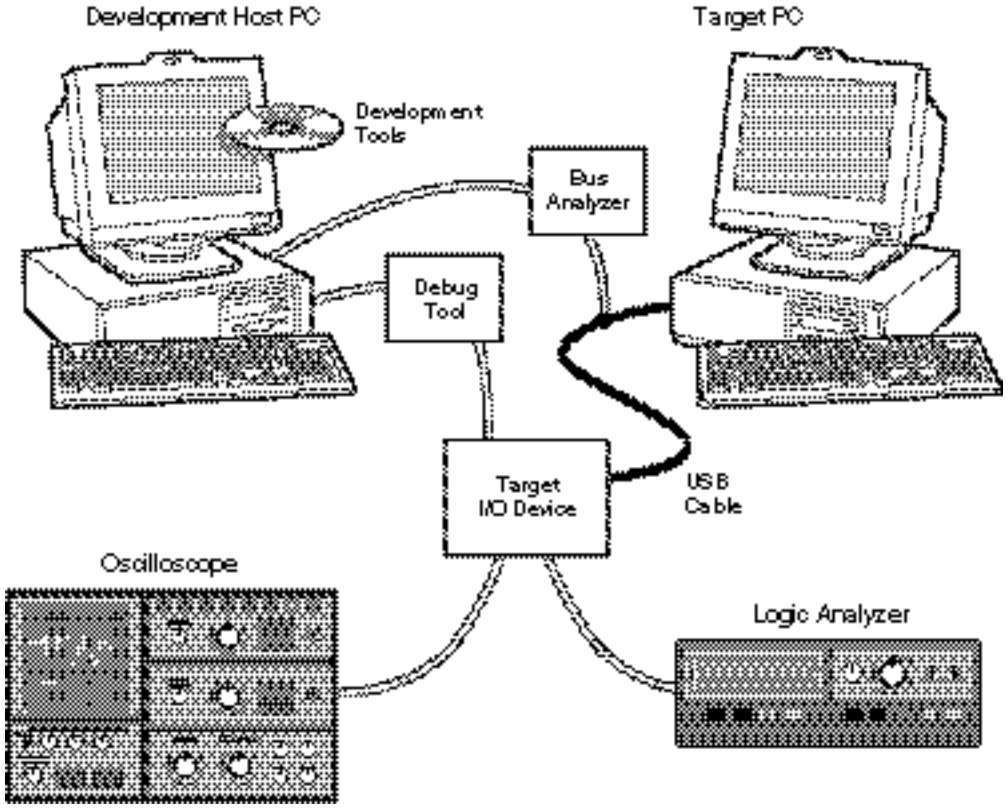


Figure 5-2. Ideal development environment

The logic analyzer and oscilloscope are used to capture signals on the target board. The bus analyzer captures all of the packets on the target I/O device's USB cable. The target PC runs the enumeration and then the application for our target board. Having a separate PC gives us flexibility and control during the debug process—if the target PC were to hang during development (a typical situation during code debugging), we still have control via the development host. The debug tool will vary with different groups of solutions, but its role of connecting the development PC host to the target is unchanged: firmware developed on the development host can be downloaded into the target and its execution controlled and observed by the debug tool. Figure 5-3 shows the development environment that will be used for the examples in later Chapters.

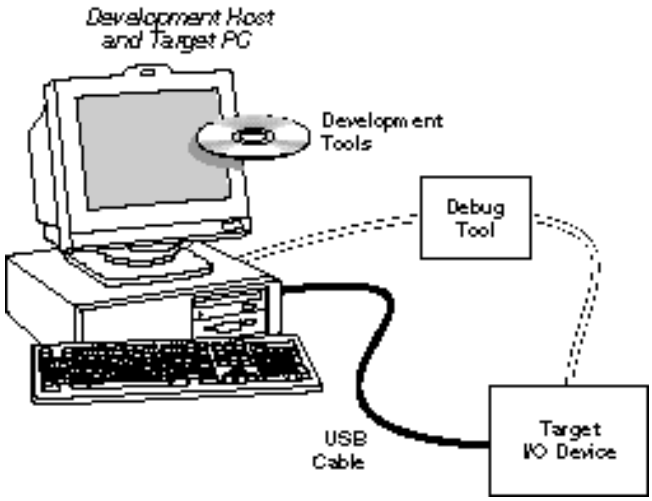


Figure 5-3. Adequate development environment for early examples

The examples start by being relatively easy; they are simple enough that neither a logic analyzer nor an oscilloscope is required. Only a limited number of USB packets will be used, so a bus analyzer is not essential. Finally, the risk of a system crash is small with our examples because the examples are predebugged; thus, it is acceptable to use a single PC for both tasks. (To generate the examples, I used the equipment shown in Figure 5-2.) Once we move beyond the initial examples, however, we'll need more development tools. First we'll add a bus analyzer (for a range of available equipment, see the Chapter 5/bus analyzer directory on the CD-ROM). The software environment is similar for all target implementations. Before discussing the firmware development and debug environment, let's look at the different development hardware that is available.

TARGET IMPLEMENTATIONS

The USB I/O device can be implemented in three different ways (Figure 5-4).

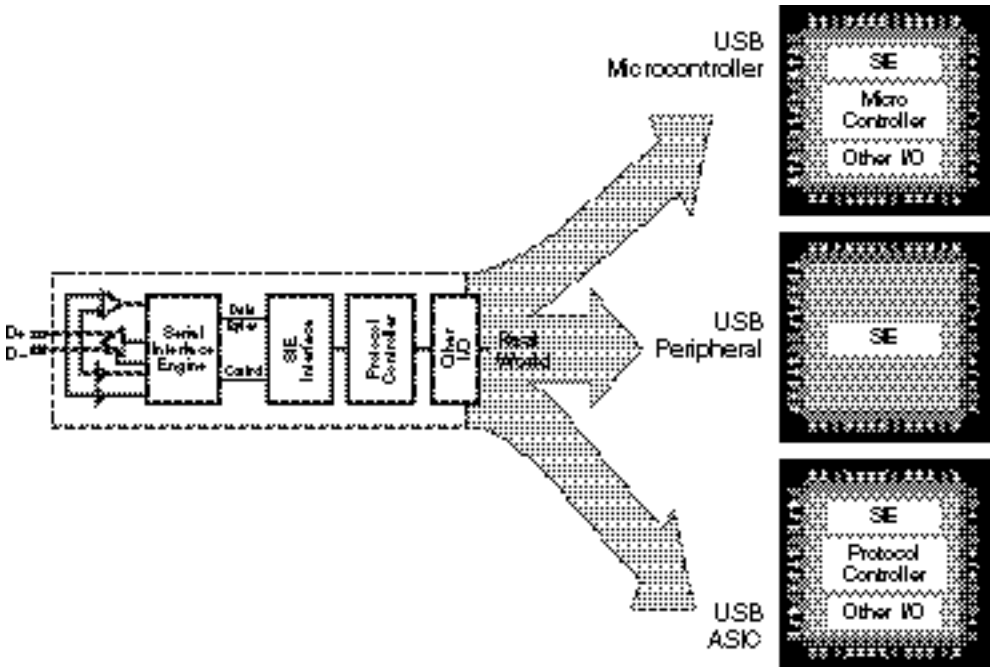


Figure 5-4. USB I/O device implementation options

The **USB microcontroller** integrates the SIE and a microcontroller onto a single component. The microcontroller contains data memory that may be expandable off-chip. The microcontroller may also contain program memory enabling a single chip solution to be implemented. Program memory may be expanded off-chip in some implementations. Two general development approaches are taken depending on the expansion capabilities of the microcontroller:

- The **"bond-out"** solution for single chip microcontrollers
- The **"debug monitor"** solution for external program memory microcontrollers

The **USB peripheral** contains the SIE and associated endpoint FIFOs in a single component. The protocol engine is implemented by an attached processor that could be any processor! The attached processor views the USB peripheral the same as other peripherals, such as a serial port or a network controller: The USB peripheral has status and control registers, and the peripheral is either the source

or sink of data. From a development perspective, this solution is developed and debugged using tools targeted for the attached processor.

The **USB ASIC** integrates the SIE, controller, and all required I/O into a single chip. This integrated chip results in the lowest product cost for the USB I/O device but the highest development cost. The development environment is similar to the bond-out solution except that different code generation tools are used on the development PC.

A Bond-out Example

A microcontroller with embedded program memory and only I/O signals on its pins is almost impossible to debug because of the limitations on observing its operation. To enable easy design and debug of these single-chip solutions, most manufacturers provide bond-out versions of their microcontrollers (Figure 5-5).

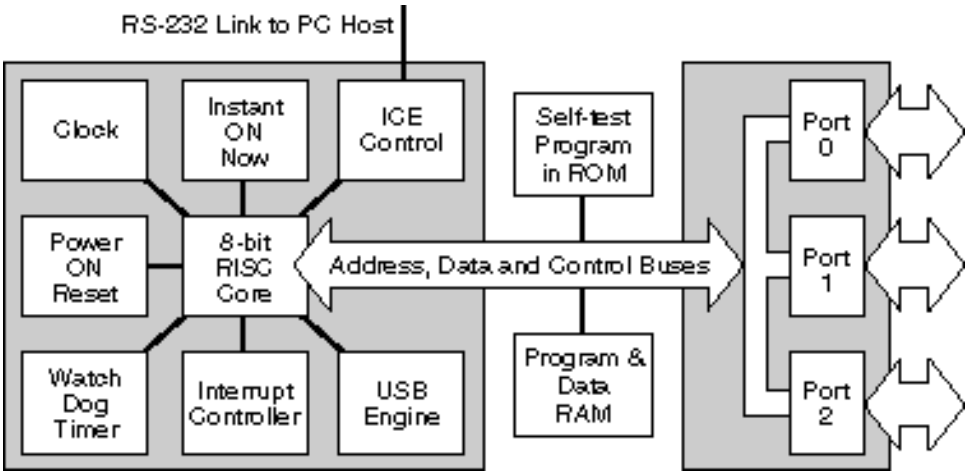
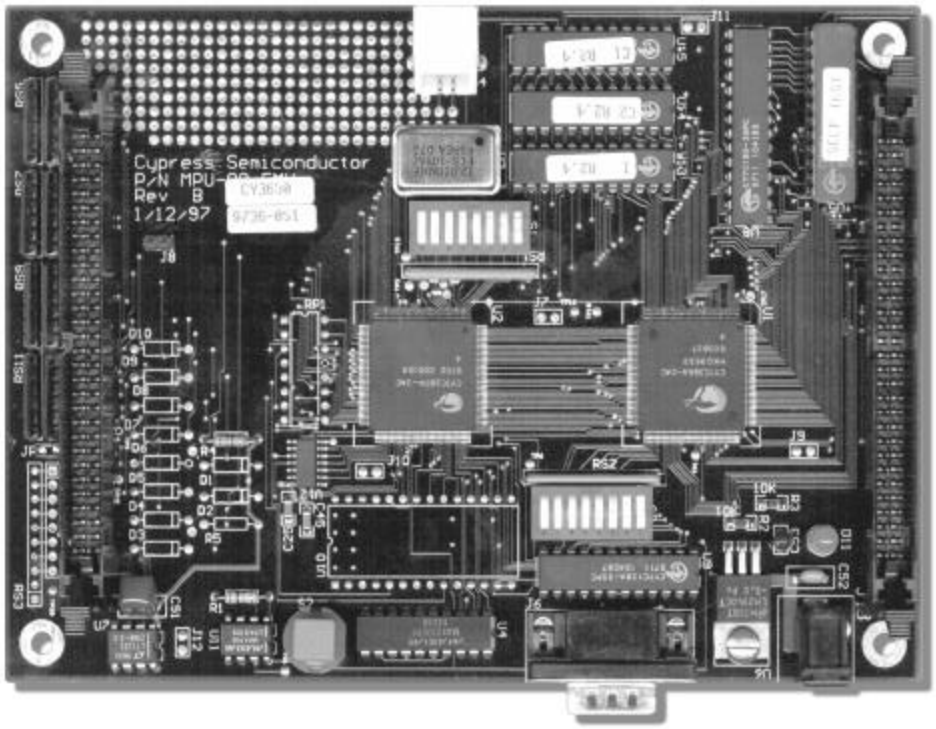


Figure 5-5. Bond-out part needs a bigger package

The development component is referred to as “bond-out” because extra internal signals not found on the pins of a production part are bonded-out to pins in a larger package. These signals give increased visibility into the operation of the microcontroller. Special control circuitry can also be added to single-step the program, halt it at specific address or data accesses, and trace operation. During the debug phase, the program is stored in RAM memory because it is writable at execution speeds—we do not need to program and erase

EPROMs to edit the program. The added capabilities and large packages of a bond-out part mean that it is a lot more expensive than the production part, but, of course, we need only one.

Figure 5-6 shows a representative bond-out development kit. This unit from Cypress Semiconductor includes a cable that operates just like a production microcontroller as far as the target I/O board is concerned.



Courtesy of Cypress Semiconductor Corp.

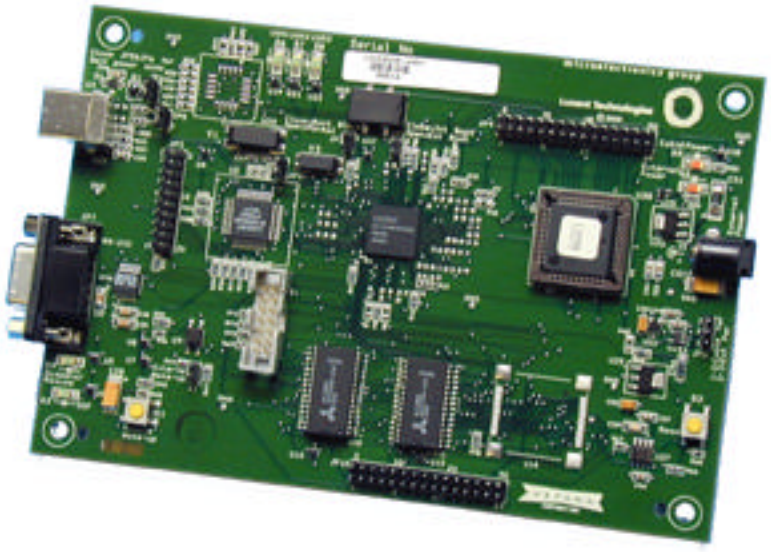
Figure 5-6. Example of debug board using a bond-out component

Figure 5-7 shows a more sophisticated tool that includes real-time trace memory. A bond-out chip is still used and additional control circuitry is added. This equipment is called an In Circuit Emulator (ICE) and includes a cable that will operate identically to the production microcontroller. The example in Figure 5-7 is from Mitsubishi and is targeted for their M37641 USB microcontroller – many manufacturers make similar products targeted at a particular microcontroller. All units include a comprehensive PC host graphical user interface that controls the operation of the microcontroller and displays its registers, memory and trace history.

Figure 5-7. In Circuit Emulator tool.

A Debug Monitor Example

If the microcontroller executes from external program memory, it is straightforward to build a debug monitor development tool. The example in Figure 5-8 is a Lucent ARM development board that includes more program memory, both EPROM-based and RAM-based. Following power-up, the microcontroller executes from the on-board EPROM where a monitor program is stored. The monitor program is used to download a program via the serial port into RAM and then is used to control its execution. The benefit of this approach is that there is no difference between the component used for development and the one used for production.



Courtesy of Lucent

Figure 5-8. Example of debug monitor environment

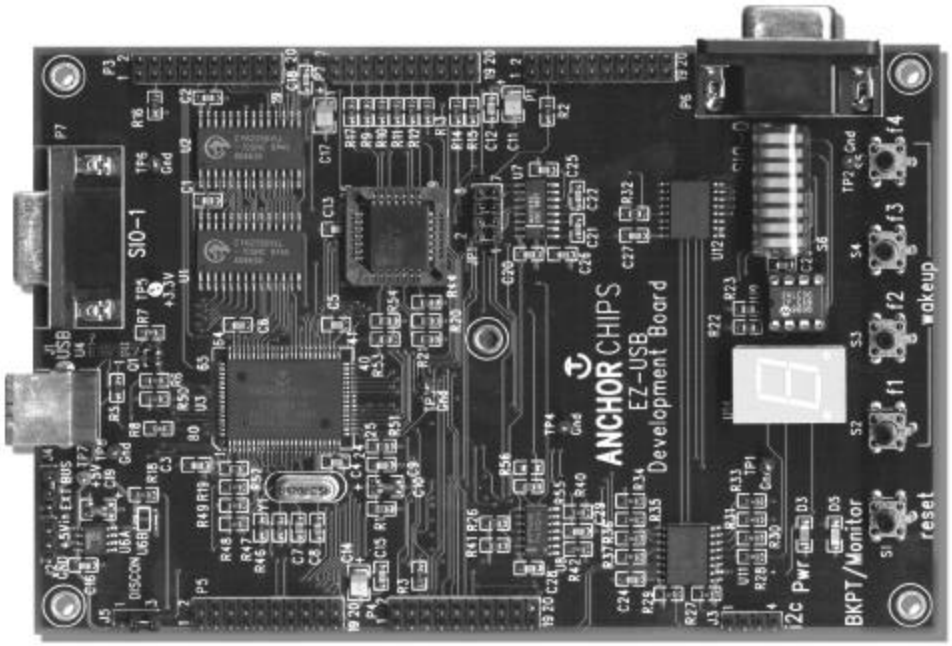
An Integrated Debug Monitor Example

An innovative approach to the debug monitor solution has been taken by Cypress Semiconductor with their EZ-USB product line.

The EZ-USB product is designed to download a program via the USB cable during enumeration. The development board in Figure 5-9 downloads a debug monitor during enumeration. To download the target program, you can use either a Windows application program or the debug monitor. The development environment is fully symbolic and includes a Keil C compiler as well as the assembler and linker toolkit.

The development board contains more hardware for test applications and debugging but uses exactly the same microcontroller device that will be used in production.

Cypress has a USB 2.0 version of this product – it looks almost identical except it uses an EZ-FX2 microcontroller.



Courtesy of Cypress Semiconductor

Figure 5-9. The EZ family has soft-loadable program memory

USB Peripheral Example

A USB peripheral design could use the same tools as the USB microcontroller examples. You have more scope since you can choose tools for the specific microcontroller/microprocessor/DSP that you are using in your target system.

Netchip created a rapid prototyping tool for their range of USB peripherals – the example in Figure 5-10 is their USB 2.0 product, the Net2290. When working with a USB peripheral, you need an I/O processor, memory and access to development tools. Netchip solved this need in an elegant way; they created a PCI-based, carrier board onto which you plug your selected USB peripheral. You use the PC host processor as your I/O device controller and you develop your software using the robust PC development environment.

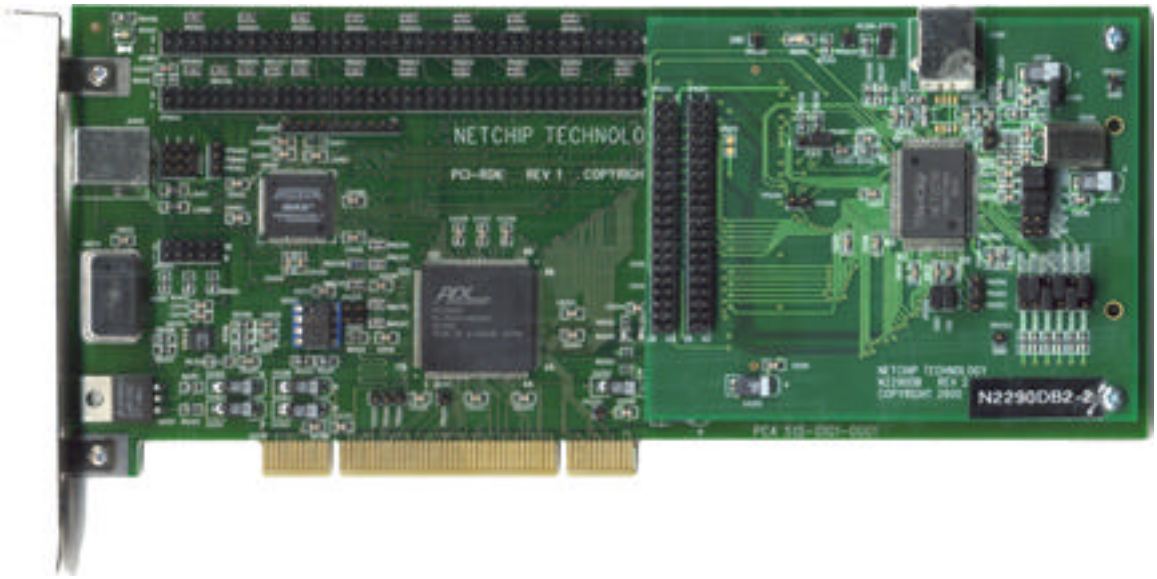


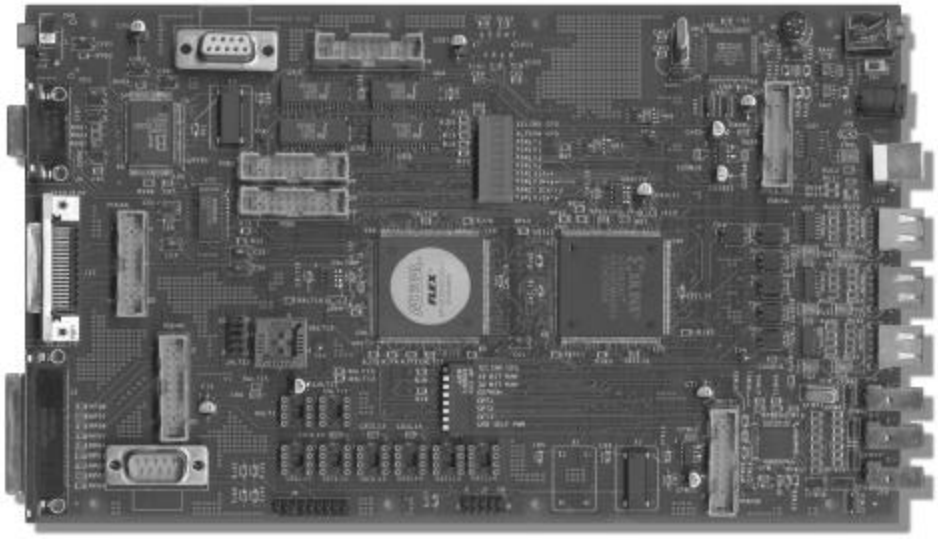
Figure 5-10. Using a PC host as an I/O controller.

Netchip also provides a suite of tools to get your project off to a fast start – this includes the basic USB enumeration and bulk loopback test. Your software development can start immediately while your hardware design is being tuned for your specific implementation. You later port your software from the PC host environment to the I/O device target environment.

USB ASIC Example

There is a lot more development work to do with the ASIC approach to USB I/O device design, because all the component hardware must also be created. However, this approach gives the designer much more freedom in the design, which can be tuned exactly to the requirements of the I/O device. The initial hardware design consists of configuring a "sea-of-gates" ASIC with the required elements of the design. VAutomation's solution has many prepackaged building blocks, such as the SIE, device and host controller, digital phase lock loop circuits, and a microcontroller interface. VAutomation also has a full 8-bit RISC microcontroller available as an ASIC library element to include in a design.

Figure 5-11 shows the development board, which includes hardware interfaces to audio, video, and parallel and serial ports. VAutomation includes example programs that exercise all of these interfaces.



Courtesy of VAutomation, Inc.

Figure 5-11. USB ASIC development board

FIRMWARE DEVELOPMENT TOOLS

Firmware for the I/O device microcontroller will be created on the development PC host. Tools specific for the microcontroller architecture will be used to write the code that delivers the responses to the PC host's requests and that implements the real-world I/O interactions. For many of the examples in this book, I have used assembler code for the microcontroller because there is a large amount of bit-handling and direct I/O manipulation. Once the complexity increases however, a C compiler will allow you to focus on the algorithms that the I/O device needs to implement. A C example is included in Chapter 9. C compilers are also available for several microcontrollers; for a medium-to-large I/O design, you should first consider using C because the code will be easier to modify and maintain.

Most microcontroller vendors supply example code that implements the enumeration stage of a design. In Chapter 7, we'll work through an example to see how the I/O device microcontroller interfaces to the Serial Interface Engine. Once you have written (or obtained) this enumeration code, you can reuse it on all projects, because the enumeration code does not change the personality of an I/O device; instead, such change is caused by entries in the descriptor tables. The run-time code will also change, of course, because different real-world interfaces will be used. In a USB I/O device project, you can take advantage of having a large opportunity to write **modules** of code—the microcontroller code behind an Interface Descriptor will be reusable on all designs that use the same descriptor.

Once the firmware is written, it is assembled/compiled and an executable module is created. This module must be downloaded into the target system for debug. All of the development systems described in this chapter allow this module to be downloaded into RAM for the debug process—we don't have to program an EPROM yet!

A debug monitor program will already be resident in the target system—this will be PROM or has been downloaded before our target program. When creating the target program, ensure that it does not use any of the memory space or I/O resources of the debug monitor program.

The debug monitor program will be used to control execution of our target firmware program. The debug monitor program also allows memory, microcontroller registers, and I/O ports to be displayed and modified. The target program can be single-stepped or run with breakpoints. The debug monitor sets a breakpoint by substituting a "trap" instruction in place of the actual instruction at the breakpoint address. When the microcontroller reaches this address, it switches from executing the target program back into executing the debug

monitor program so that variables can be analyzed and program execution studied.

An In-Circuit emulator includes a real-time trace facility that captures the execution address of the microcontroller for later analysis. This trace tells you exactly where the microcontroller has been. The trace can be stopped when the microcontroller accesses certain memory addresses, typically those it should not be addressing anyway, and the trace is studied to understand where the microcontroller "took a wrong turn." A trace tool, or attached logic analyzer with similar characteristics, is invaluable when you are trying to track down program bugs that send the microcontroller off-course.

In Chapter 7 we'll see that most of the protocol controller program runs under interrupt control. While this interrupt-driven approach is the most efficient method to use, it can create bugs that are hard to track down if we are not precise with interrupt handling. More care must be taken on microcontrollers that support multiple levels of interrupt nesting—a stack overflow is a common bug in such programs and is hard to track down.

In the TOOLS directory on the CD-ROM, I have included a set of tools for the MCS 51 microcontroller. These tools were kindly provided by Keil Corporation, and they can be used to re-create all of the MCS 51 examples in the next chapter. These free tools are 100 percent functional but have been modified so they can create a target program only up to 4 KB long. This is ample for our examples. A set of tools that supports programs greater than 4 KB is available directly from Keil. The Keil tools on the CD-ROM also incorporate a sophisticated debug monitor program called Dscope that allows full symbolic access to all of the variables and memory locations—this is a lot easier than dealing with hexadecimal numbers.

PC HOST DEVELOPMENT TOOLS

In sharp contrast to the I/O intensive programming of the I/O device, the PC Host software does not access any hardware at all! The USB host controller is a shared resource in the PC and, as such, needs to manage the accesses from multiple programs. In the Windows WDM implementation a device driver called USBD.SYS is used to make USB requests. All programs call USBD.SYS, which will schedule the USB transactions on your behalf. Figure 5-12 shows the layered implementation of USB support in Windows 98.

Figure 5-12. USB software layers in the Windows OS family

An application program does not call USBD.SYS directly. Device drivers call USBD.SYS with kernel mode privileges. We could write a device driver (this is covered in Chapter 10) or we could use a pre-supplied device driver. Initially we shall use the Human Interface Device class driver (HIDUSB.SYS) to gain access to USB resources – an applications program makes calls into a system library. This API (applications program interface) is the topic of Chapter 7.

There are a variety of PC host based tools that can help with our I/O device development process.

USB-SOFTWARE TOOLS

A variety of tools is available, and included on the CD-ROM, to ease the development of the target I/O board.

USB View

USB View uses the data tables and USB routines of the operating system to draw a table of attached USB devices (Figure 5-13). The device descriptors are read and displayed.

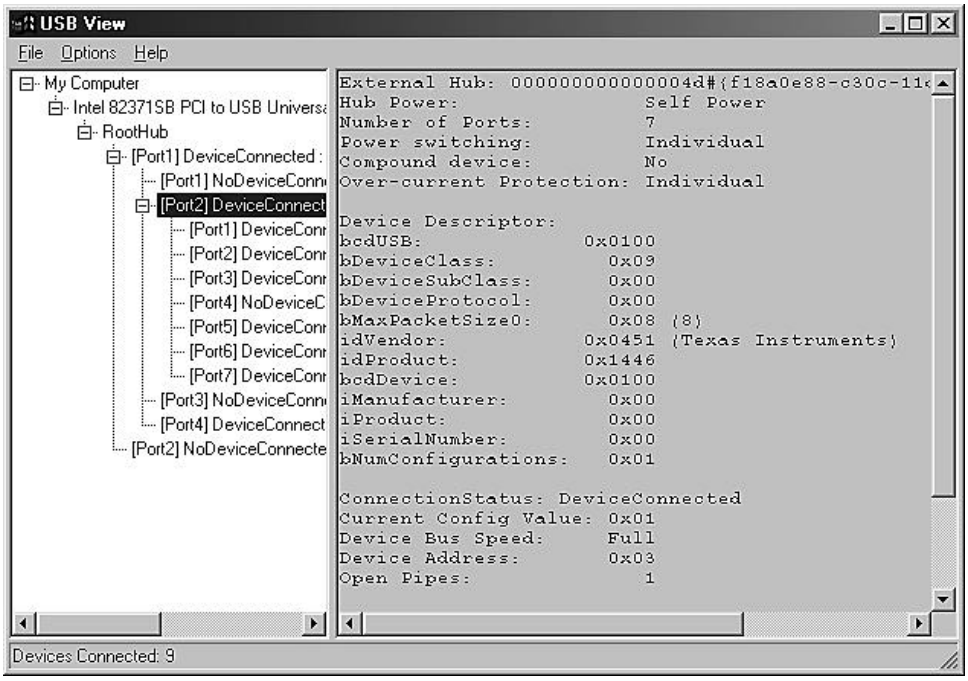


Figure 5-13. Operating system's view of USB devices

Hidview

Starting Hidview puts the operating system USB software into diagnostic mode for newly attached HID devices. The tool can then observe and monitor HID reports (Figure 5-14).

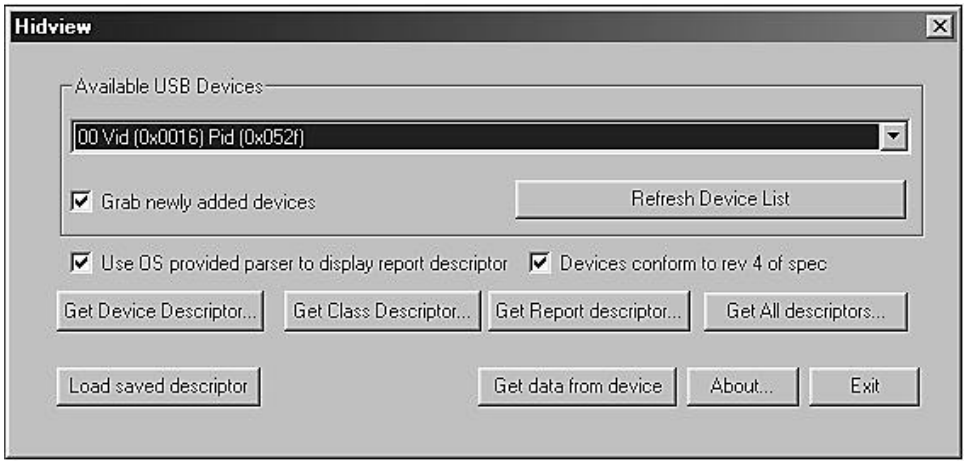


Figure 5-14. Debug of HID device using Hidview

HID Table Creator

A HID **report** is a detailed, compacted data structure. The HID table creator is an interactive program that guides you through creating a report descriptor (Figure 5-15). You can save the output in multiple formats that can be pasted into an assembler file or used as a C declaration file. This tool saves a lot of time, since it can check the report descriptor for errors or omissions. A report descriptor must be checked with this tool and, if there are no errors, it may be used in your I/O device.

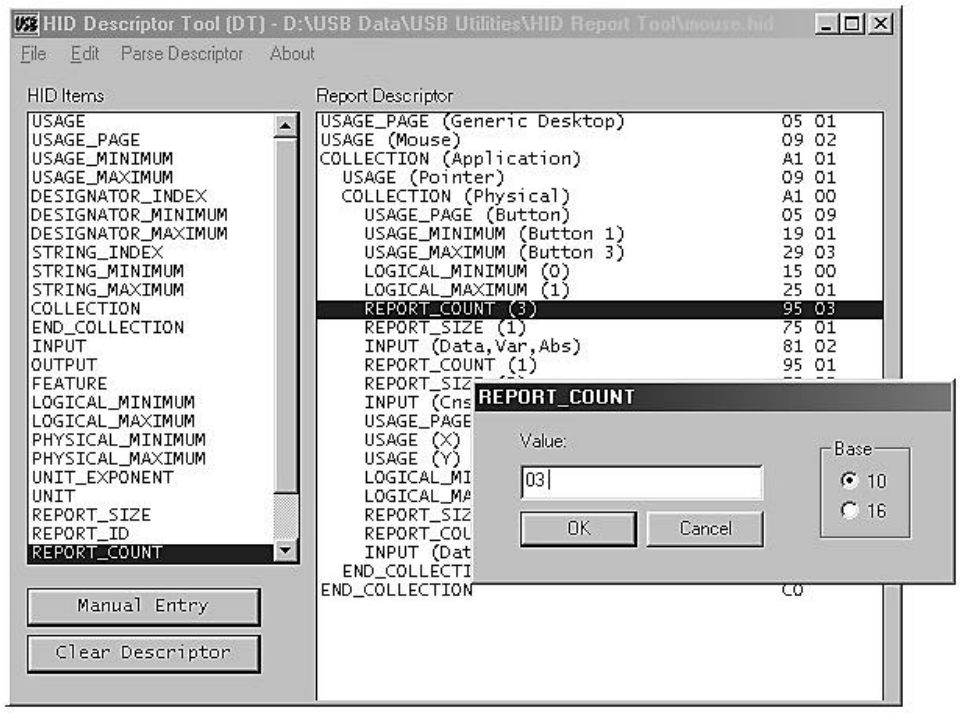


Figure 5-15. Use HID tool to create report descriptors

SINGLE STEPPING OF USB ENUMERATION

The enumeration phase, when controlled by the operating system, is very fast. Any descriptor errors are handled poorly by the operating system and it is easy to create meaningless registry entries that will slow down your development. A development tool, called the Peripheral Development Kit (PDK) is available from www.usb.org - it consists of a USB 2.0 host controller board (Figure 5-16), a Windows 2000 driver stack (beta version) and a Single Step Application program (Figure 5-17).

Install the PCI-based, USB 2.0 host controller into your development PC host and install the drivers from Microsoft. At the time of writing only Win 2000 was supported – check www.usb.org for newer drivers. No other software on your development PC will know about this special PCI card so the OS will ignore it. The Single Step program has exclusive access to this card and can control its operation exactly.

Start the single step application program and attach your I/O device under test to the host controller card that we just inserted. You can now single step through the enumeration process and ensure that your I/O device is responding correctly.

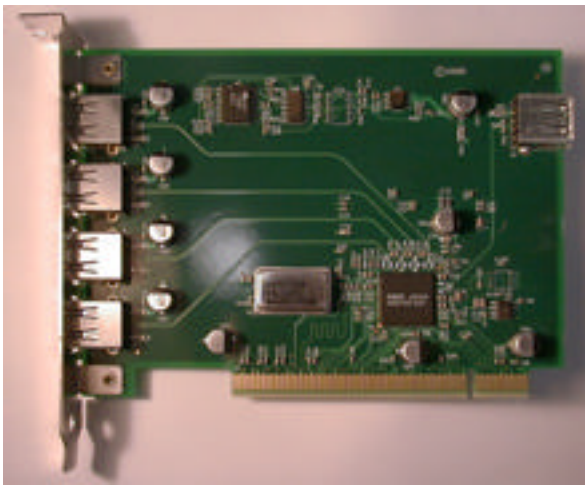


Figure 5-16. USB 2.0 host controller

<<paste screen shot here>>

Figure 5-17. Single Step application program

USB HARDWARE TOOLS

It is interesting and educational to use an oscilloscope to look at the differential signals on the USB data wires, but this is not a productive tool to use in debugging a system. What we need is an observation tool that understands the packetized nature of the bus and can display the bus transactions at a higher level.

Figure 5-18 shows a USB probe from Crescent Software that is used in conjunction with a Tektronix logic analyzer. The figure shows their USB-2XP product that samples at 2.4GHz (5x over sample rate) to capture 480Mb/s bus transactions, they also have an USB-XLC product that samples a full/low-speed bus. Both products include a hierarchical display of event, packet, transaction and control transfer bus activity, with device class decoding. A typical trace sequence is shown in Figure 5-19.

Figure 5-18. High performance bus probe

<<paste screen shot here>>

Figure 5-19. Event display and time coordinated bus display

Computer Access Technology Corporation has been working with USB since its inception, and their insight into the design challenges of a USB subsystem is obvious in their CATC tools (Figure 5-1).

Courtesy of Computer Access Technology Corp.

Figure 5-1. USB bus-specific tools from CATC

The CATC tools range from their Detective for USB observation to the Chief that can analyze, capture, replay, and run tests. All the tools capture activity on the USB bus and display it in the form of packet diagrams (Figure 5-20). Each token packet is displayed in a different color, and the sequence of packets that make up a transaction are grouped together.

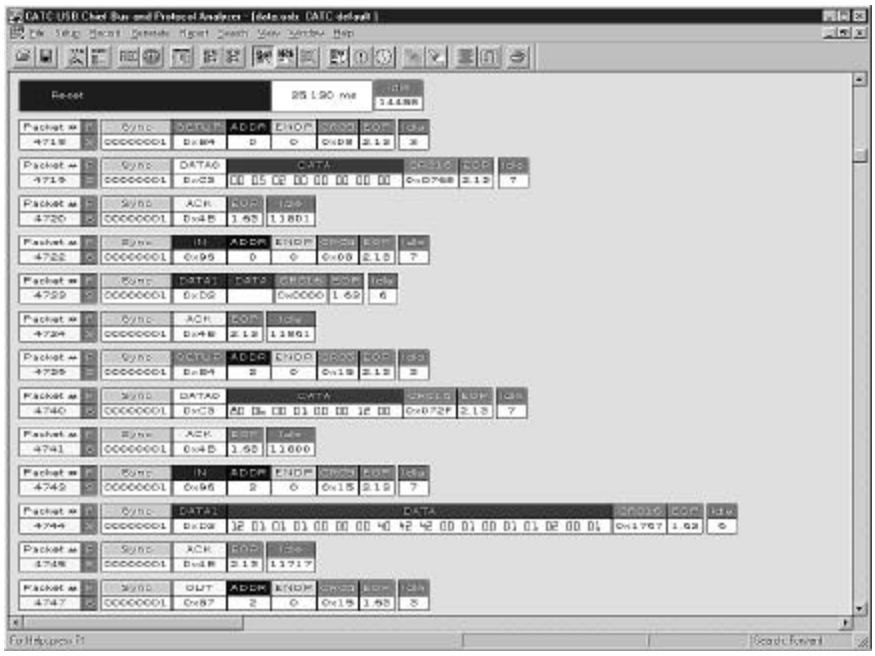


Figure 5-20 USB packets displayed as packets

<<paste screen shot here>>

Figure 5-21 Looking at both sides of a high-speed hub

All of the CATC tools capture bus activity for later analysis. The use of color to display different packet types and the grouping of building-block packets into transactions allow the designer to quickly interpret what has happened on the bus. The simpler CATC tools capture all bus activity while the more elaborate tools have programmable capture and programmable triggers. Being able to isolate packets sent to a specific device or triggering after, for example, device 42 has received 58 DATA0 packets, aids the debugging of more sophisticated USB devices. The high-end CATC tools can also be used to generate bus traffic for device reliability testing and failure analysis.

<< Discuss other tools here >>

CHAPTER SUMMARY

The development environment for a USB I/O device is mature. Different tools are available that match the target solution for the I/O device. These tools are available from multiple vendors and support their microcontroller solutions very well. Third-party tools are also available to support development efforts, and I have included a suite of these tools on the CD-ROM. I've also provided several host-based programs that will help in the development of your I/O device.